26 bit Selectors on IA32

Greg Law & Julie McCann,
Dept. of Computing,
City University, London, UK
email: {gel,jam}@soi.city.ac.uk

Abstract

The Intel 80386 (IA32) segmentation scheme is well suited to implementing memory protection in the new generation of finely-grained component-based operating systems. However, IA32 segments are identified by 14 bit selectors, giving a maximum of just 16,382 segments addressable at a time. This is contrary to the goal of fine-grained protection — a large system comprised of fine-grained components may require many times this number of components be addressable. Changing the IA32 specification to allow 32 bit selectors would be ideal, but is clearly impractical. This paper presents the strategy adopted in the Go! *component-based operating systems, which is to* virtualise *selectors, using the IA32 'segment-descriptor table' as a* cache *of a larger table, indexed by 26 bit selectors.*

1 Introduction

It is a common misconception that the IA32 segmentation scheme is simply a hangover from the 8086 'so-called' segmentation, and that any operating system worth its salt will reject the segmentation-model in favour of paging; segmentation can provide a flexible and powerful means of protection (especially when *combined* with paging). Recent operating systems such as L4 [2], QNX [3] and Go! [5] use the Intel's segmentation to provide lightweight protection. While it is true that using IA32's segmentation does hinder an operating system's portability, source-code portability of operating systems has been identified as a major performance bottleneck [1, 2] (note that applications can remain blissfully unaware that they are protected using segmentation).

Segments have the obvious advantage over pages that they may have an arbitrary size, and can therefore avoid the internal fragmentation associated with paging. Memory lost to internal fragmentation does not present a problem on conventional, process-based systems, where a few dozen protection-domains are active at a time. However, with the fine-grained protection required by the new generation of component-based operating systems, the (relatively large) fixed-sized nature of pages begins to limit the granularity at which protection may be employed with reasonable memory usage. The fact that 'memory is cheap' does not solve this problem, particularly for embedded systems (which are much more sensitive to small increases in 'unit price' than desktop ones). However, using segments to enable lightweight protection on the Intel immediately runs into a problem: fine-grained protection implies many protected address-spaces, yet IA32 is limited to just 16,382 segments.

The remainder of this paper is structured as follows: Section 2 introduces the memory protection scheme employed by IA32 and Section 3 explains how Go! component identifiers (effectively segment selectors) were extended from 14 bits to 26. Section 4 presents some experiments we conducted to test this scheme, and Section 5 concludes.

2 Memory Protection on IA32

This section introduces the IA32's (relatively complicated) memory protection scheme. Any preconceptions of the 8086 segmentation scheme should be discarded: the 8086 used segmentation

by name only — this 'hack' is replaced by a much more useful segmentation model as soon as the processor is placed into 'protected mode'.

IA32 combines segmentation and paging orthogonally, as introduced in MONADS [4]. Here, an address generated by an instruction (known as the logical address), is converted into a linear address using the segmentation mechanism. If paging is enabled, the linear address is then fed through the paging hardware to generate a physical address. To explain the segmentation model we will first assume that paging is turned off.

Like a page, a segment is a region of memory with a *translation* and *access rights*. However, unlike pages, segments have a variable size, usually with byte or word granularity. The segment's translation is made up of its *base* address, and a *limit*. The access rights denote the ways in which a segment may be accessed (namely: execute, read-only data or read/write data). Every memory access is checked so that it is (a) within the access-rights, and (b) the logical address is not greater than the segment's limit. If these conditions are not met, a 'general protection fault' (GPF) is generated. Otherwise, the logical address is added to the segment's base address to generate the linear memory address at which the access is made.

The processor maintains a set of currently 'active' segments, with different segments for code, data and stack — all code is loaded via the currently active code segment (that is, the instruction pointer is always relative to the code-segment), static data accesses go via the current data segment, and stack operations go via the current stack segment.

Each segment's access-rights and translation are held in a data structure known as a *descriptor* (analogous to a page-table-entry on paged systems). The set of all segments in the system is described by the *Global Descriptor Table* (GDT) (analogous to a page-table). An individual segment is identified by its index into this table, known as a *selector*. This arrangement is demonstrated in Figure 1.

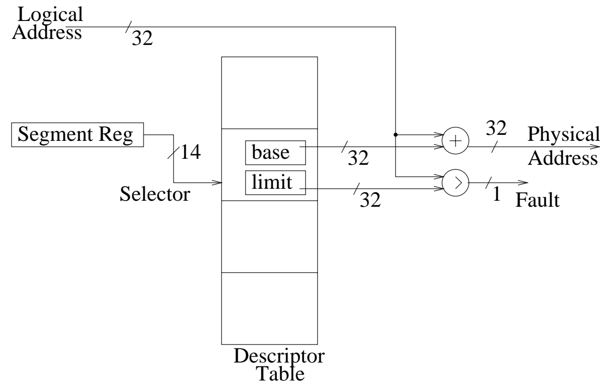

Figure 1: Memory Protection with Segmentation

The IA32 passes linear addresses through a conventional 2-level page table to produce a physical address. Combining segmentation and paging like this gives the potential for powerful and flexible architectures.

As with paging, context-switches can be effected by changing the descriptor-table so that outgoing segments are inaccessible, and in-coming ones accessible. As well as the GDT, the Intel allows the definition of several *Local* Descriptor Tables (LDTs), one of which may be active at a time (a bit in the selector specifies whether to use the GDT or LDT)[1].

Many chip designers have abandoned segmentation, partly because the addition required by segmentation for each memory access is considerably more work than the concatenation required by paging. This potentially hampers a segmented architecture's performance — naively implemented

[1]A system may have multiple LDTs, but only can be active at a time. Also, multiple LDTs are not suited to the 'single-address-space' object model of systems like Go!

segmentation results in architectures with either longer cycles, or extra pipeline stages. Logical-address cache indexing might resolve these problems, although the authors are unaware of any architectures that do this.

3 26 bit Component Identifiers

As we have seen in Section 2, the segment selectors on IA32 are only 14 bit, giving a maximum of 16,382 segments are addressable at any time. This is an unacceptable limitation if, for example, a protection mechanism requires one segment per component in a finely-grained, component-based system. Go! is one such system.

The solution employed by Go! is to associate a 32 bit *reference* (of which only the bottom 26 bits are significant[2]) with each component, known as an *ObjRef*. This reference is independent of the component's data segment selector. This is possible because IA32 is limited to a few thousand segment *selectors*, but there is no such limit on the number of *descriptors* (2^{52} distinct descriptors are available, not including different access permissions). 64 bit ObjRefs were considered, but rejected because the benefits (simpler large-scale distribution) were not seen to outweigh the costs (more complicated and expensive communication in the local case).

The GDT becomes in effect a *cache* of up to 8,191 segment descriptors (LDTs are not used in the current implementation). The Go! ORB (Go!'s equivalent of a kernel) maintains a table known as the Component Descriptor Table (CDT). The CDT maps ObjRefs to selectors: the relevant selector if the given ObjRef is valid and cached in the GDT; an invalid selector otherwise. The entries indicating that an ObjRef is uncached or invalid are chosen so that attempting to load a segment register with a segment not cached in the GDT, results in a GPF. The ORB responds to faults triggered by ObjRefs not cached in the GDT by loading the relevant ObjRef into the GDT, rejecting other entries as necessary. Faults due to the attempted use of an invalid ObjRef (rather than a valid but uncached one) result in the error being communicated to the calling component. Using GPFs to catch the attempted use of invalid and uncached ObjRefs avoids potentially expensive conditional branches in the ORB's critical paths.

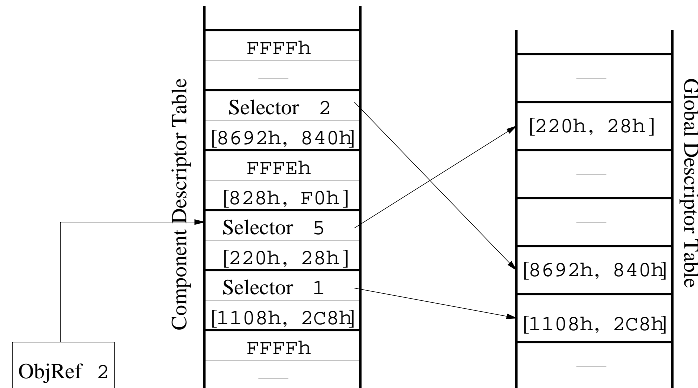

Figure 2: 32 bit ObjRefs with 14 bit selectors

Figure 2 shows an example GDT and CDT, and an example translation for ObjRef `2`. ObjRef `2`'s selector field in the CDT is `5`, indicating that it is cached in the GDT, at entry `5`. The diagram also shows that 2 other segments exist, with ObjRefs `1` and `4`. ObjRefs `0` and `5` are invalid, since their selector fields in the CDT are `FFFF`h. ObjRef `3` is valid, but not cached in the GDT (hence its selector field in the CDT is `FFFE`h). Note that the component's data segment descriptor from

[2]Only 26 bits of the 32 bit reference are significant because the reference is an index into a table, and so must be shifted left before use.

the GDT is duplicated in the CDT. This allows the descriptor to be loaded into the GDT when an an ObjRef not cached in the GDT is referenced.

In the above example, when ObjRef `3` is referenced, the ORB will attempt to load the data segment register with `FFFE`h (ObjRef `3`'s selector field in the CDT). This will trigger a GPF, which will be received by whatever component is responsible for interrupt management. On noticing that the fault occurred in the ORB's code-segment (selector `8`) the interrupt management component must call the ORB's `fault` routine. In response, the ORB copies the component's data segment descriptor from the CDT to a free entry in the GDT. If there are no free entries, one is rejected (currently using a 'random' rejection policy). Note that invalid entries in the CDT shown in Figure 2 have their descriptor field undefined, as do unused entries in the GDT. This is because there are no circumstances under which these entries will be referenced.

4 Performance Experiments and Results

It is reasonable to assume a very high hit-ratio of components cached in an 8K slot GDT (although exact figures will depend on the application). Still it is important to ascertain the penalty for a 'cache miss' (i.e. invoking a method on a component with its data segment not cached in the GDT). It is even more important that the scheme does not adversely affect the inter-component method-call latency if a component's data segment is cached in the GDT.

All measurements were made using an Intel Pentium P54C running at 90 MHz, with 32 MB of 90 ns EDO DRAM. All results are in machine cycles, measured using the Pentium's `rdtsc` instruction.

The experiments examine the time taken to transfer control between two components using Go!'s `xfer` primitive [5]. Three experiments were conducted: one for an implementation using 14 bit selectors, and one each using 26 bit ObjRefs where the callee component's data segment was and was not cached in the GDT[3]. Timings use a hot cache and branch-prediction buffers (i.e. the experiments are run in a tight loop, and the times averaged).

All results are in cycles.

Primitive	`xfer` time (cycles)
Using 14 bit selector	31
Using ObjRef cached in GDT	29
Using ObjRef not cached in GDT	984

The experiments show that the price of a 'miss' is acceptable for most applications, but more importantly, that the scheme does not slow down the common case.

5 Summary

We have shown that the 14 bit selectors of IA32 need not limit the number of segments addressable at once. Using the limited descriptor-table as a cache of available segments, wider selectors can be used. Furthermore, employing the protection hardware so that attempted use of invalid ObjRefs triggers a protection fault, potentially expensive conditional branches can be avoided when checking whether ObjRefs are cached in the GDT (and during ObjRef verification). Avoiding conditional branches like this has demonstrated speed *improvements* over using 14 bit selectors naively.

[3]the caller component will always be in the GDT since its data segment selector is loaded into the data-segment register.

References

[1] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An operating sytem architecture for application-level resource management. In *Proc. 15th Symposium on Operating System Principles (SOSP-95)*, pages 251–266, 1995.

[2] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-Kernel-based systems. In *Proceedings of the 16th Symposium on Operating Systems*, pages 66–77, 1997.

[3] Dan Hildebrand. An architectural overview of QNX. In USENIX Association, editor, *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures: 27–28 April, 1992, Seattle, WA, USA*, pages 113–126, Berkeley, CA, USA, April 1992. USENIX.

[4] J. Keedy and J. Rosenberg. Support for objects in the MONADS architecture. In *Proc. Workshop on persistent object systems*, pages 202–213, Newcastle NSW (Australia), January 1989.

[5] Greg Law and Julie McCann. A New Protection Model for Component-Based Operating Systems. In *Proceedings of the IEEE Conference on Computing and Comminications, Phoenix, AZ, USA*, February 2000.